MOTOROLA
SEMICONDUCTOR
APPLICATION NOTE

Order this document
as AN427/D

AN427

MC68HC11 EEPROM Error Correction Algorithms in C

By Richard Soja
Motorola Ltd
East Kilbride
Glasgow

INTRODUCTION

This application note describes a technique for correcting one bit errors, and detecting two bit errors, in a block of data ranging from 1 to 11 bits in length. The technique applied is a modified version of a Hamming code, and has been implemented entirely in C. Additional functions have been provided to program and read the EEPROM on an MC68HC11 microcontroller unit using the error encoding and decoding algorithms.

ENCODING AND DECODING ALGORITHMS

Some texts [1], [2] describe the use of simultaneous equations to calculate check bits in Hamming distance-3 error correcting codes. These codes are so named because there are at least 3 bit differences between each valid code in the set of available codes. The codes are relatively easy to generate and can be used to correct one bit errors. However, their main drawback is that if two bit errors occur, then the correction will be made erroneously. This is because the condition of two bit errors corresponds exactly with a one bit error from another valid code.

The technique described here is based on an algorithmic strategy which produces Hamming distance-4 codes over the range of 1 to 11 data bits. This type of code is capable of correcting single bit errors and detecting 2 bit errors.

Alternatively, if the errors are only to be detected, without correction, then up to 3 bit errors can be detected. The reason for this is that the condition of a 3 bit error in one code corresponds to a one bit error from an adjacent valid code. The implication of this is that, if the algorithms are used to correct errors, then a 3 bit error will be corrected erroneously, and flagged as a 1 bit error.

The C program is divided into 3 modules, plus one header file:

1. EECOR1.C

   This is the main program segment, and serves only to illustrate the method of calling and checking the algorithms.

2. HAMMING.C

   This module contains the functions which encode and decode the data.

3. EEPROG.C

   This module contains the EEPROM programming functions tailored for an MC68HC11 MCU.

4. HC11REG.H

   This is the header file which contains the MC68HC11 I/O register names, defined as a C structure.

IMPLEMENTATION OF ERROR CORRECTION STRATEGY

The basic principle of decoding the error correcting codes is to use a Parity check matrix, H, to generate a syndrome word which identifies the error. The H matrix can be generated as follows:

1. Identify how many data bits are needed. For example: 8 data bits

2. Use the standard equation to derive the number of check bits required: If k is the number of check bits, and m the number of data bits, then for the Hamming bound to be satisfied:

$$2^k \geq m + k + 1$$

This document contains information on a new product. Specifications and other information herein are therefore subject to change without notice.

© MOTOROLA LTD., 1990

A simple way to understand why this equation holds true is as follows: If one can generate a check code which is able to identify where a single error occurs in a bit stream, then the check code must have at least the same number of unique combinations as there are bits in the bit stream, plus 1 extra combination to indicate that no error has occurred. e.g. if the total number of data plus check bits were 7, then the check code must consist of 3 bits, to cover the range 1 to 7 plus one extra (0) to indicate no error at all.

In this example, if m=8 then, by rearranging the above equation:

$$2^k - k - 1 >= 8$$

One way to solve for k is to just select values of k starting at say, 1 and evaluating until the bound is reached. This method is implemented algorithmically in function InitEncode() in module HAMMING.C

For m=8, the solution is k=4. Note that this value exceeds the Hamming bound, which means that additional data bits can be added to the bit stream, thus increasing the efficiency of the code. In fact, the maximum number of data bits is 11 in this case.

3. A Parity matrix, H is created from a 'horizontally orientated' binary table.The number of columns (b1 to b12) in the matrix correspond to the total number of data and check bits, and the number of rows (r1 to r4) to the number of check bits.

```
i.e. b1  b2  b3  b4  b5  b6  b7  b8  b9  b10 b11 b12
r1    1   0   1   0   1   0   1   0   1   0   1   0
r2    0   1   1   0   0   1   1   0   0   1   1   0
r3    0   0   0   1   1   1   1   0   0   0   0   1
r4    0   0   0   0   0   0   0   1   1   1   1   1
```

Because the H matrix in this form, is simply a truncated 4 bit binary table, it can easily be generated algorithmically.

4. The position of all the check bits (C1 to C4) within the encoded word is the position of the single 1s in the columns of H. The remaining bits correspond to the data bits (D1 to D8).

```
i.e. C1  C2  D1  C3  D2  D3  D4  C4  D5  D6  D7  D8
     1   0   1   0   1   0   1   0   1   0   1   0
     0   1   1   0   0   1   1   0   0   1   1   0
     0   0   0   1   1   1   1   0   0   0   0   1
     0   0   0   0   0   0   0   1   1   1   1   1
```

5. Each check bit is generated by taking each row of H in turn, and modulo-2 adding all bits with a 1 in them except the check bit positions.

```
i.e.  C1=D1+D2+D4+D5+D7
      C2=D1+D3+D4+D6+D7
      C3=D2+D3+D4+D8
      C4=D5+D6+D7+D8
```

6. The syndrome, s, is the binary weighted value of all check bits.

```
i.e.  s=1*C1+2*C2+4*C3+8*C4
```

7. The error position (i.e. column) is determined by the value of the syndrome word, provided it is not zero. A zero syndrome means no error has occurred. Note that this error correction technique can correct errors in either data or check bits - which is not necessarily the case with certain other error correction strategies.

   The advantage of this method, where the check bits are interspersed in a binary manner throughout the code word, is that the error position can be calculated algorithmically.

   An important point to note is that the parity check matrix described above generates Hamming distance-3 codes, which means that 2 errors will cause erroneous correction. This can be fixed by adding an extra parity check bit, C5, which is the modulo-2 addition of all data and check bits together.

```
i.e.C5=C1+C2+D1+C3+D2+D3+D4+C4+D5+D6+D7+D8
```

The code word then becomes:

```
C1 C2 D1 C3 D2 D3 D4 C4 D5 D6 D7 D8 C5
```

To determine if an uncorrectable error has occurred (i.e. 2 errors) in the received word, the extra parity bit is tested. If the syndrome is non-zero and the parity bit is wrong, then a correctable error has occurred. If the syndrome is non-zero and the parity bit is correct, then an uncorrectable error has occurred.

EFFICIENCY

The following table lists the relative efficiencies of this algorithm, against data size.

Data bits	Encoded bits	Efficiency %
1	4	25
2	6	33
3	7	43
4	8	50
5	10	50
6	11	55
7	12	58
8	13	62
9	14	64
10	15	67
11	16	69

The implementation of the above techniques are given in the module HAMMING.C.

In order to maintain orthogonality in the EEPROM algorithms, the encoded data used by the functions in module EEPROG.C are forced to either 1 byte or 2 byte (word) sizes. This also eliminates the complexities of packing and unpacking data in partially filled bytes.

CONCLUSIONS

In this application note, the encoding algorithm's generator matrix is the same as the parity check matrix.

The C functions <read> and <write> in the module HAMMING.C return a status value - 0, 1 or 2 - which indicates whether the data has no errors, 1 corrected error, or 2 erroneously corrected errors. This means that if the status value is 0 or 1, then the data can be assumed good. If the status value is 2, then the data will be bad.

Alternatively the functions can be used for error detection only, without correction. In this case, a status value of 1 corresponds to either 1 or 3 bit errors, while a status value of 2 indicates that 2 bit errors have occurred.

By using the C functions listed in this application note, the encoded data size can easily be changed dynamically. To do this, the function <InitEncode> must be called with the required new data size. The global variables used by all the encoding, decoding and EEPROM programming and reading functions are automatically updated. This allows the encoding and error correction process to be virtually transparent to the user. In addition, the functions <write> and <read> will automatically increment the address pointer by the correct encoded data size set up by <InitEncode>. This simplifies the structure of loops to program and read back data. Example code is provided in module EECOR1.C.

The encoding and decoding algorithms listed here may be applied to other forms of data, such as that used in serial communications, or for parallel data transfers.

By incorporating the error correction or detection-only schemes described in this application note, the integrity of data storage and transfer can be greatly improved. The impact on EEPROM usage is to increase its effective reliability and extend its useful life beyond the manufacturers' guaranteed specifications.

REFERENCES

[1] Carlson, 'Communication Systems', Chapter 9, McGraw-Hill.

[2] Harman, 'Principles of the Statistical Theory of Communication', Chapter 5, McGraw-Hill

MODULE EECOR1.C

```
Tests EEPROM error detection using a modified hamming encoding scheme.

typedef unsigned char byte;
typedef unsigned int word;


/* Global variables used by main() */
 byte *ee_addr,*start_addr,*end_addr,i,Error;
 word data;

/*************************************************************************/

/* External global variables */
 extern byte CodeSize;                          /* = number of bits in encoded data */

/* External Functions */
 extern byte read(word *data,byte **addr);      /* Function returns error status */
 extern byte write(word data,byte **addr);      /* "
/* Table of Status returned by read and write functions
            Returned Status                     Condition
                 0                  No errors detected or corrected.
                 1                  One error detected and corrected.
                 2                  Two errors detected, but correction is erroneous.

Notes:
1/  When the returned value is 2, the function <read> will returned a bad value in variable
<data> due to the inability to correctly correct two errors. <read> also automatically increments
the address pointer passed to it, to the next memory space. The incremented value takes into
account the actual size of the encoded data. i.e. either 1 or 2 byte increment.

2/  Function <write> also performs a read to update and return an error status. This gives an
immediate indication of whether the write was successful. <write> also automatically increments
the address pointer passed to it, to the next free memory space. The incremented value takes into
account the actual size of the encoded data. i.e. either 1 or 2 byte increment.

*/

/*************************************************************************/

int main()
{
  CodeSize=InitEncode(11);              /* Get code size (less 1) needed          */
                                        /* by 11 data bits                        */

  ee_addr=(byte *)0xb600;               /* Initialise EEPROM start address        */
  for(i=1;i<=0x10;i++)                  /* and 'erase' EEPROM                     */
      Error=write(0x7ff,&ee_addr);      /* Function successful if Error<>2        */

  ee_addr=(byte *)0xb600;               /* Reset EEPROM address                   */

  Error=write(0x5aa,&ee_addr);          /* Write 0x5aa & increment ee_addr        */
  Error=write(0x255,&ee_addr);          /* Write 0x255 at next available address  */

  CodeSize=InitEncode(4);               /* Change number of data bits to 4        */
  start_addr=ee_addr;                   /* Save start address for this data       */

  for(i=1;i<0x10;i<<=1)                 /* Program 'walking 1s'                   */
      Error=write(i,&ee_addr);
  end_addr=ee_addr;                     /* Save end address                       */

  ee_addr=start_addr;
  while (ee_addr<end_addr)              /* Read back all the 4 bit data           */
      Error=read(&data,&ee_addr);       /* <data> good if Error=0 or 1            */

} /* main */
```


MODULE HAMMING.C

```
/*Modules to Generate hamming codes of distance 4, for data sizes in the range 1 bit to 11 bits.
The upper bound is limited by the encoded word type bit range (16 bits).

Corrects 1 bit error in any position (check or data), and detects 2 bit errors in any position.

After execution of the <Decode> function, the global variable <ErrFlag> is updated to indicate
level of error correction.

i.e.   ErrFlag        Condition
          0       No errors detected or corrected.
          1       One error detected and corrected.
          2       Two errors detected, but correction is erroneous.

Note that when ErrFlag is 2, function <Decode> will return a bad value, due to its inability to
correctly correct two errors.
*/

#define TRUE 1
#define FALSE 0
typedef unsigned char byte;
typedef unsigned int word;

 byte DataSize,CodeSize,EncodedWord,ErrFlag;

/* Function prototypes */
 byte OddParity(word Code);
 word Power2(byte e);
 byte InitEncode(byte DataLength);
 word MakeCheck(word Data);
 word Encode(word Data);
 word Decode(word Code);


byte OddParity(Code)
word Code;
/*
Returns TRUE if Code is odd parity, otherwise returns FALSE
*/

{
  byte p;

  p=TRUE;
  while (Code!=0)
  {
    if (Code & 1) p=!p;
    Code>>=1;
  }
  return(p);
}

word Power2(e)
byte e;
/*
 Returns 2^e
*/

{
 word P2;
 signed char i;

 P2=1;
 if ((signed char)(e)<0)
    return(0);
 else
```

```
 {
    for (i=1;i<=(signed char)(e);i++)
        P2<<=1;
    return(P2);
 }
}

byte InitEncode(DataLength)
byte DataLength;
/*
 Returns the minimum number of total bits needed to provide
 Hamming distance 3 codes from a data size defined by passed
 variable <DataLength>. This value also updates global variable <DataSize>.
 i.e. finds the minimum solution of (k+m) for the inequality:
                  2^k ≥ k + m + 1

 In addition, updates global variable <EncodedSize> to reflect number of bytes
 per encoded data. <EncodedSize> will be either 0 or 1.
*/


{
  byte CheckLength,i;

  DataSize=DataLength;  /* DataSize used by other functions in this module */
  CheckLength=1;
  while ((Power2(CheckLength)-CheckLength-1)<DataLength)
        CheckLength++;
  i=CheckLength+DataLength;
  EncodedWord=i / 8;                    /* =0 if byte sized, =1 if word sized          */
  return(CheckLength+DataLength);
}

word MakeCheck(Data)
word Data;
/*
 Returns a check word for Data, based on global variables <DataSize>
 and <CheckSize>. The H parity matrix is generated by a simple for loop.
*/

{
  byte i,H,CheckSize,CheckValue,Check,CheckMask;
  word DataMask;

  Check=0;
  CheckMask=1;
  CheckSize=CodeSize-DataSize;
  for (i=1;i<=CheckSize;i++)
  {
      CheckValue=FALSE;
      DataMask=1;
      for (H=1;H<=CodeSize;H++)
      {
          if ((0x8000 % H)!=0)          /* Column with single bit set                  */
          {
             if ((H & CheckMask)!=0)
                 CheckValue^=((DataMask & Data)!=0);
             DataMask<<=1;
          }
      }
      if (CheckValue) Check|=CheckMask;
      CheckMask<<=1;
  }
  return(Check);
}
```



```
word Encode(Data)
word Data;
/*
 Returns an encoded word, consisting of the check bits
 concatenated on to the most significant bit of <Data>.
 A single odd parity bit is concatenated on to the Encoded word to
 increase the hamming bound from 3 to 4, and provide 2 bit error
 detection as well as 1 bit correction.
 Uses global variables <Datasize> and <CodeSize> to determine the
 concatenating positions.
*/

{
  word Code;

  Code=Data | (MakeCheck(Data)<<DataSize);
  if (OddParity(Code))
     Code|=Power2(CodeSize);
  return(Code);
}

word Decode(Code)
word Code;
/*
 Returns the error corrected data word, decoded from <Code>.
 Uses global variable <DataSize> to determine position of the
 check bits in <Code>.
 Updates global variable <ErrFlag> to indicate error status i.e.:
        ErrFlag          Status
           0          No errors found
           1          Single error corrected
           2          Double error - invalid correction
*/

{
  word ParityBit,Data,Check,ErrorCheck,Syndrome,DataMask;
  byte DataPos,CheckSize,CheckPos,H,DataBit;

  ErrFlag=0;
  ParityBit=Code & Power2(CodeSize);            /* Extract parity bit.                 */
  DataMask=Power2(DataSize)-1;                  /* Make data mask */
  Data=Code & DataMask;                         /* Extract data bits.                  */
  CheckSize=CodeSize-DataSize;                  /* Extract check bits,                 */
  Check=(Code>>DataSize) & (Power2(CheckSize)-1); /* ignoring parity.                  */
  ErrorCheck=MakeCheck(Data);
  Syndrome=Check ^ ErrorCheck;                  /* Get bit position of error.          */
  if (Syndrome>0) ErrFlag++;                    /* Increment flag if error exists.     */
  H=0;
  DataPos=0;
  CheckPos=DataSize;
  DataBit=TRUE;

  while ((H!=Syndrome) & (DataPos<DataSize))    /* Identify which data or              */
  {
    H++;                                        /* code bit is in error.               */
    DataBit=(0x8000 % H);
    if (DataBit) DataPos++;
    else         CheckPos++;
  }
  if (DataBit) Code^=Power2(DataPos-1);
  else         Code^=Power2(CheckPos-1);
  Code|=ParityBit;
  if (OddParity(Code)) ErrFlag++;
  return(Code & DataMask);
}
```

MODULE EEPROG.C

```
/*Module to program MC68HC11 EEPROM.

 Contains <read> and <write> functions to encode and decode data
 formatted by modified hamming scheme.

*/

#include <HC11REG.H>
#define regbase (*(struct HC11IO *) 0x1000)

#define eras 0x16
#define writ 0x02
typedef unsigned char byte;
typedef unsigned int word;

 union twobytes
 {
  word w;
  byte b[2];                                  /* Word stored as MSB,LSB          */
 } udata;

 extern byte EncodedWord,ErrFlag;

/* Function prototypes */

 extern word Encode(word Data);
 extern word Decode(word Code);

 void delay(word count);
 void eeprog(byte val,byte byt,byte *addr,word count);
 void program(byte byt,byte *addr);
 byte read(word *data,byte **addr);
 byte write(word data,byte **addr);

void delay(count)
word count;
{
  regbase.TOC1=regbase.TCNT+count;            /* Set timeout period on OC1 and   */
  regbase.TFLG1=0x80;                         /* clear any pending OC1 flag.     */
  do;while ((regbase.TFLG1 & 0x80)==0);       /* Wait for timeout flag.          */
}


void eeprog(val,byt,addr,count)
byte val;                                     /* val determines Erase or Write operation  */
byte byt;                                     /* byt is byte to be programmed             */
byte *addr;                                   /* addr is address of encoded byte in EEPROM */
word count;                                   /* count is number of E clock delays        */
{
  regbase.PPROG=val;                          /* Enable address/data latches     */
  *addr=byt;                                  /* Write value to required eeprom location  */
  ++regbase.PPROG;                            /* Enable voltage pump             */
  if (count<100) count=100;                   /* Allow for software overhead     */
  delay(count);                               /* wait a bit                      */
  --regbase.PPROG;                            /* Disable pump,then addr/data latches      */
  regbase.PPROG=0;
}
```



```
void program(byt,addr)
byte byt;
byte *addr;
{
  eeprog(eras,byt,addr,20000);                /* First erase byte                */
  eeprog(writ,byt,addr,20000);                /* Then write value                */
}

byte read(data,addr)
word *data;
byte **addr;
{
  udata.b[1]=*(*addr)++;                      /* Read back data LSB first, and inc address */
  if (EncodedWord)                            /* If word stored then read MSB    */
     udata.b[0]=*(*addr)++;                   /* Inc address for next call to this function */
  else                                        /* else only byte stored, so clear MSB      */
     udata.b[0]=0;
  *data=Decode(udata.w);                      /* Decode data, which updates <ErrFlag>,    */
  return(ErrFlag);                            /* and return ErrFlag              */
}

byte write(data,addr)
word data;
byte **addr;
{
  byte *oldaddr;

  udata.w=Encode(data);                       /* Encode data.                    */
  oldaddr=*addr;                              /* Save initial address for verification.   */
  program(udata.b[1],(*addr)++);              /* Program LSB first to allow for either    */
  if (EncodedWord)                            /* 1 or 2 byte encoded data        */
     program(udata.b[0],(*addr)++);           /* MSB of word sized data,& inc address     */
  return(read(&udata.w,&oldaddr));            /* Return <ErrFlag> to calling segment      */
}
```

HC11REG.H

```
/*      HC11 structure - I/O registers for MC68HC11 */

struct HC11IO {
 unsigned char   PORTA;                 /* Port A - 3 input only, 5 output only     */
 unsigned char   Reserved;
 unsigned char   PIOC;                  /* Parallel I/O control                     */
 unsigned char   PORTC;                 /* Port C */
 unsigned char   PORTB;                 /* Port B - Output only                     */
 unsigned char   PORTCL;                /* Alternate port C latch                   */
 unsigned char   Reserved1;
 unsigned char   DDRC;                  /* Data direction for port C                */
 unsigned char   PORTD;                 /* Port D                                   */
 unsigned char   DDRD;                  /* Data direction for port D                */
 unsigned char   PORTE;                 /* Port E                                   */

/*      Timer Section */

 unsigned char   CFORC;                 /* Compare force                            */
 unsigned char   OC1M;                  /* Oc1 mask                                 */
 unsigned char   OC1D;                  /* Oc1 data                                 */
 int             TCNT;                  /* Timer counter                            */
 int             TIC1;                  /* Input capture 1                          */
 int             TIC2;                  /* Input capture 2                          */
 int             TIC3;                  /* Input capture 3                          */
 int             TOC1;                  /* Output compare 1                         */
 int             TOC2;                  /* Output compare 2                         */
 int             TOC3;                  /* Output compare 3                         */
 int             TOC4;                  /* Output compare 4                         */
 int             TOC5;                  /* Output compare 5                         */
 unsigned char   TCTL1;                 /* Timer control register 1                 */
 unsigned char   TCTL2;                 /* Timer control register 2                 */
 unsigned char   TMSK1;                 /* Main timer interrupt mask 1              */
 unsigned char   TFLG1;                 /* Main timer interrupt flag 1              */
 unsigned char   TMSK2;                 /* Main timer interrupt mask 2              */
 unsigned char   TFLG2;                 /* Main timer interrupt flag 2              */

/*      Pulse Accumulator Timer Control */

 unsigned char   PACTL;                 /* Pulse Acc control                        */
 unsigned char   PACNT;                 /* Pulse Acc count                          */
```



```
/*      SPI registers */

 unsigned char   SPCR;                  /* SPI control register                     */
 unsigned char   SPSR;                  /* SPI status  register                     */
 unsigned char   SPDR;                  /* SPI data    register                     */

/*      SCI registers */

 unsigned char   BAUD;                  /* SCI baud rate control                    */
 unsigned char   SCCR1;                 /* SCI control register 1                   */
 unsigned char   SCCR2;                 /* SCI control register 2                   */
 unsigned char   SCSR;                  /* SCI status  register                     */
 unsigned char   SCDR;                  /* SCI data    register                     */

/*      A to D registers */

 unsigned char   ADCTL;                 /* AD control register                      */
 unsigned char   ADR[4];                /* Array of AD result registers             */

/*      Define each result register */

#define          adr1    ADR[0]
#define          adr2    ADR[1]
#define          adr3    ADR[2]
#define          adr4    ADR[3]

 unsigned char   Rsrv[4];               /* Reserved for A to D expansion            */

/*      System Configuration */

 unsigned char   OPTION;                /* System configuration options             */
 unsigned char   COPRST;                /* Arm/Reset COP timer circuitry            */
 unsigned char   PPROG;                 /* EEPROM programming control reg           */
 unsigned char   HPRIO;                 /* Highest priority i-bit int & misc        */
 unsigned char   INIT;                  /* RAM - I/O mapping register               */
 unsigned char   TEST1;                 /* Factory TEST control register            */
 unsigned char   CONFIG;                /* EEPROM cell - COP,ROM,& EEPROM en        */

 };

/*      End of structure HC11 */
```